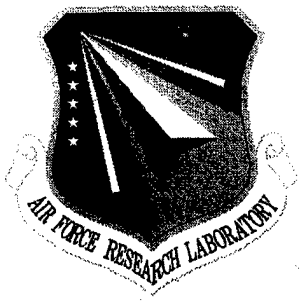
AFRL-IF-RS-TR-2001-14
Final Technical Report
February 2001



A TECHNOLOGY INVESTIGATION SUPPORTING SOFTWARE ARCHITECTURE AND ANALYSIS FOR EVOLUTION

Carnegie Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E095

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-14 has been reviewed and is approved for publication.

APPROVED: Roy F. Stratton

ROY F. STRATTON
Project Engineer

FOR THE DIRECTOR: James A. Collins

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

A TECHNOLOGY INVESTIGATION SUPPORTING SOFTWARE ARCHITECTURE AND ANALYSIS FOR EVOLUTION

David Garlan,
Mary Shaw, and
Jeannette Wing

Contractor: Carnegie Mellon University
Contract Number: F30602-97-2-0031
Effective Date of Contract: 12 February 1997
Contract Expiration Date: 30 September 2000
Short Title of Work: A Technology Investigation Supporting Software Architecture and Analysis For Evolution
Period of Work Covered: Feb 97 - Sep 00

Principal Investigator: David Garlan
Phone: (412) 268-5056
AFRL Project Engineer: Roy F. Stratton
Phone: (315) 330-3004

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Roy F. Stratton, AFRL/IFTD, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	FEBRUARY 2001	Final Feb 97 - Sep 00

4. TITLE AND SUBTITLE
A TECHNOLOGY INVESTIGATION SUPPORTING SOFTWARE ARCHITECTURE AND ANALYSIS FOR EVOLUTION

5. FUNDING NUMBERS
C - F30602-97-2-0031
PE - 62702F
PR - E095
TA - 01
WU - 01

6. AUTHOR(S)
David Garlan, Mary Shaw, and Jeannette Wing

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Carnegie Mellon University
School of Computer Science
5000 Forbes Road
Pittsburgh PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203

Air Force Research Laboratory/IFTD
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2001-14

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Roy F. Stratton/IFTD/(315) 330-3004

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In this report we summarize the approaches and result of the project "A Technology Investigation Supporting Software Architecture and Analysis for Evolution," carried out at Carnegie Mellon University under funding from the DARPA Evolutionary Design of Complex Systems Program. in this project we addressed problems in managing the evolution of complex software by providing new technology to describe and analyze a system's software architecture. This report summarizes our efforts in two areas: (1) understanding an existing systems in preparation for making changes, and (2) actually making the changes to the system correctly. We also describe the application of our techniques and tools to industrial and defense architecture.

14. SUBJECT TERMS
Software Architecture, program Analysis, Program generation, Design Environments, Architecture Description Languages, Software Evolution, Software Design

15. NUMBER OF PAGES
44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures.

1. Introduction

Software design is a creative and complex task, where many decisions and trade-offs are made in order to produce a solution that satisfies software requirements. Unfortunately, the information used and created in the design process is often produced in an *ad hoc* manner and not communicated effectively to subsequent phases of software development. This becomes especially problematic when the time comes to make changes to the software. Analysis of the impact of a proposed change, and whether the change violates design assumptions or warrants revisiting trade-offs, are difficult to ascertain. This means that evolving software often introduces new errors, or the software suffers from attrition. In order to provide a more principled approach to evolving or changing software, support for defining, analyzing and communicating the design of the system is crucial.

The high level design structure of a system can be defined using software architectures. A software architecture captures the organization of a system as a collection of interacting components. A well-defined architecture can help a designer to reason about properties at a high level of abstraction. Architectural design has always played a strong role in the success of complex software systems. Unfortunately, these architectural descriptions are often informal or transient; discussed by designers but not communicated beyond the design process. This results in several problems in regard to software evolution:

- *Architectures are often poorly understood by developers.* This means that the implementation of a software architecture may not be consistent with the design that was intended in the design process.
- *Design choices are not based on solid software engineering principles.* The choices made rely on past experience of the designers involved, and may not be applicable to the system being designed.
- *Architectural designs cannot be analyzed to determine consistency or completeness.* Because the notations used are not defined, there are no methods for being confident that the design is internally consistent or does not have portions undefined.
- *Architectures cannot be enforced as the software evolves.* It is difficult or impossible to ascertain whether a change violates critical assumptions made by the designers, and may interfere with other seemingly unrelated parts of the system.

In addition to the above problems, there is scant tool support for architects to help them in their tasks. All of these problems result in a large cost when it comes to changing a system.

1.1. Innovative Claims

In our grant proposal, we argued that we could reduce the cost of evolving software systems by providing new technology that improves our ability (1) to understand an existing system in preparation for making changes, and (2) to actually make the changes to the system correctly.

During the course of this grant, we have followed this two-pronged approach. Specifically:

- We have provided new capabilities for understanding a system in order to make changes through
  - notations and tools that capture the designers' intentions about system software architecture and preserve them as part of the final system;
  - a conceptual vocabulary for expressing these intentions;
  - improved system-level analysis tools to determine properties of a given system and the impact of a proposed change.
- We have improved our capability for correctly making modifications by providing
  - mechanisms to specify architectural design constraints that determine what changes can be made to a system without violating its integrity assumptions;
  - techniques to detect and resolve architectural mismatch in heterogeneous compositions;
  - tools that ensure correspondence between an architectural design and the implementation that it represents.

The remainder of this report is organized into four sections. Section 2 describes the tools and techniques that have been applied to address our architectural approach to understanding systems in order to make changes. In Section 3, we discuss improved capabilities for making correct modifications to systems. Various projects in which our approaches have been used are discussed in Section 4, which also forms our evaluation and technology transfer. Finally, Section 5 provides our conclusions and future work.

2. Understanding software systems

For the correct evolution of software systems, it is necessary for a designer's intent to be captured so that the guidelines and decisions made at design time can be communicated to those changing the system. Design decisions are critical in understanding the trade-offs between various possible designs, and in communicating critical decision criteria that should be considered during a change. This means that these decisions can be followed in a change or at least modified with deliberate intent. To achieve this, we have developed tools and techniques in two thrusts. The first thrust was to be able develop various notations that can be used to capture software architectures and to provide a vocabulary that can be used to communicate design intentions. We have developed various prototype design tools to aid designers in composing their designs using this notation. These notations and tools are the focus of Section 2.1.

Notations for capturing software architectures are not all that is required for understanding a system's design. It is also desirable to be able to use these notations to perform analysis of the system, both to satisfy designers that a design is correct with respect to certain attributes, and also to ensure that modifications to a design are correct. Expressing these analyses also helps to capture designers' intentions and providing tools

to automate these analyses helps to check that these intentions are met in a design. Section 2.2 discusses our approaches.

2.1. Design notations

Within the scope of this project, various design notations have been developed and used at CMU to describe various aspects of software architectures. These notations range in their generality and application. For example, we took a leading role in the development of Acme, a style-independent architecture description language (ADL), which can be used both as an architectural interchange language, as well as an ADL in its own right. On the other hand, UniCon is an architectural description language whose focus is on supporting the variety of architectural parts and styles found in the real world and on constructing systems from their architecture descriptions, and so is closer to implementation. Ajax is a set of tools for providing sophisticated static analysis of Java programs. Our work on composable connectors investigates methods for applying various transformations to connectors to make them more sophisticated by adding, for example, security or dependability capabilities to architectures.

Our work on tools led to the development of a graphical design environment called AcmeStudio, a graphical design environment that uses Acme as its design notation. It allows designers to draw designs using components and connectors and store and retrieve them as Acme design documents. Our work on the ADL Workbench facilitates the integration of a number of existing design tools within the EDCS program in a methodical manner. UniCon is an architecture tool that has the ability to generate code from architectural diagrams, while Ajax allows code level analyses that can help make code level changes more reliable.

2.1.1. Acme and AcmeStudio

Acme [12], [23] is an ADL that was initially designed as an architecture interchange format but has evolved into a full-fledged ADL. Many existing ADLs are domain- or style-specific, in the sense that they are designed to express software architectures for particular domains using a vocabulary specific to that domain. Examples of such style-specific ADLs are C2, which is an event-based style, and Meta-H [3], which is applied to real-time avionics control systems. The Acme language comprises two categories of specification that can be used to specify an architecture:

- The structural elements of an architecture consist of generic *components*, *connectors* and *attachments*. Components have *ports*, which represent their interfaces, and connectors define *roles*, representing their interfaces. Attachments associate ports and roles, thereby defining a *system* that represents an instance of an architecture as a graph of components and connectors. All structural elements may have associated properties that can represent domain- or tool-specific information. Furthermore, hierarchy can be expressed using representations, which may further decompose any of the structural elements into more detail, or may represent alternative views or sub-architectures.

Figure 1 describes a simple system architecture defining two components, split and sort, each defining two ports, stdin and stdout. Each of these defines one property, throughput. Also defined is one connector that defines two roles, source and sink. The structure of the system is defined by the attachments, which define the connections between the components.

```
system demo = {
  component split = {
    ports {stdin; stdout}
    property throughput : int = 25;
  }
  component sort = {
    ports {stdin; stdout}
    property throughput : int = 40;
  }
  connector pipe = {
    roles {source; sink}
  }
  attachments {
    split.stdout to pipe.source;
    sort.stdin to pipe.sink;
  }
}
```

Figure 1. Example Acme system definition.

```
family pipeAndFilter = {
  component type filterT = {
    ports {stdin; stdout};
    property throughput : int;
  }
  connector type pipeT = {
    roles {source; sink};
  }
}

system pfdemo = {
  component spilt : filterT = new filterT with {
  property throughput = 25;
  }
  component smooth : filterT = new filterT with {
    property throughput = 40;
  }
  connector pipe : pipeT = new pipeT;
  attachments {
    split.stdout to pipe.source;
    split.stdin to pipe.sink;
  }
}
```

Figure 2. Using Acme families to describe systems.

- The typed elements allow a designer to define a style-specific vocabulary that can be used in a particular design. In Acme, these styles are called *families*. Families contain type definitions of components and connectors, which may include required ports or roles, or properties. When using these in a system definition, they provide a vocabulary that may be specific to certain domains.

  For example, Figure 2 illustrates the system described in Figure 1, but utilizing a family to define a design vocabulary. The family pipeAndFilter family defines two types: filterT and pipeT. Each of these types describes a structure that a designer will mean when they use the pipeAndFilter style. In the system definition (pfdemo), instances of these types are used. When the types for the components are instantiated, a value is given to the throughput property. The typing mechanism allows designers to use pre-specified architectural elements when constructing a software architecture. These types represent the vocabulary of a particular style.

AcmeStudio is a design environment running on the Microsoft Windows platforms that provides a graphical front-end to building Acme descriptions. The graphical interface frees the designer from having to specify Acme syntax and allows users to create all of the Acme constructs and perform various analyses in a more intuitive diagrammatic environment. Figure 3 shows a screenshot of the current AcmeStudio interface, picturing a diagram of the Acme description from Figure 2. The screenshot consists of five areas with which a designer may interact.

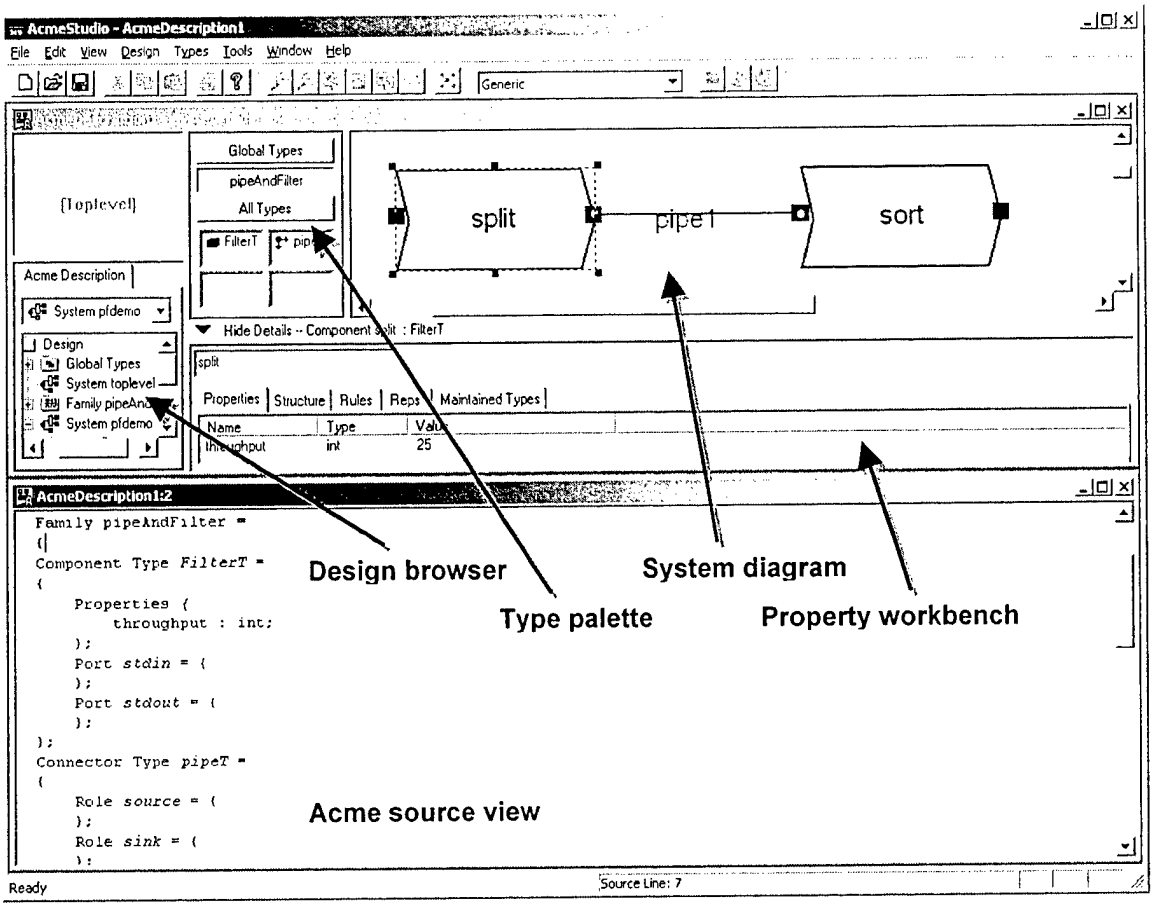


Figure 3. AcmeStudio screenshot

- **Design browser**. A hierarchical depiction of all the elements of an Acme design is shown in this window. Individual elements can be shown in more detail and this view can be used to navigate through the diagram.
- **Type palette**. All the types associated with a particular family are shown in this window. Currently, the system being defined is of type pipeAndFilter, and so those types are shown in this palette. It is possible to navigate to the diagrammatic definition of these types using this window.
- **System diagram**. This window shows a graphical depiction of the architecture being described. Users may drag types from the type palette to instantiate them in the diagram or use menus to achieve the same result. Attachments are formed by dragging roles (attached to connectors as lines) over ports in components.
- **Property workbench**. Properties can be defined and viewed using the property workbench, which shows more details of the selected component. In addition to properties, a designer may view the representations, types, and rules (discussed in Section 3.1).

- **Acme source browser**. Although it is unnecessary for a designer in this environment to interact with the Acme language, AcmeStudio allows the Acme source to be displayed. Editing in this view is not supported.

AcmeStudio also allows integration with various other tools, and facilitates design analysis using these tools. Currently, there is support for analyzing Armani constraints and conducting performance analysis based on queuing theory.

2.1.2. UniCon

UniCon is an architecture description language organized around two symmetrical constructs: A system is composed from identifiable *components* of various types that interact via *connectors* in distinct, identifiable ways. Components are specified by *interfaces*; they correspond roughly to compilation units of conventional programming languages and other user-level objects (e.g., files). Connectors are specified by *protocols*; they mediate interactions among components. That is, they define the rules governing component interaction and specify any auxiliary implementation mechanisms required. Connectors do not in general correspond directly to compilation units; they are realized as table entries, linker instructions, dynamic data structures, system calls, initialization parameters, utility servers, and so on. An architectural style is based on selected types of components and connectors, together with rules about other properties of the system, such as connection topology.

Our purpose is to support the abstractions used in practice by software designers. We developed and distributed a prototype implementation. This implementation provides a testbed for experiments with a variety of system construction mechanisms.

- it distinguishes among different types of components and different ways these components can interact.
- it supports abstract interactions such as data flow and scheduling on the same footing as simple procedure call.
- it produces running code including "glue" required for connectors, thus making the architectural description the definitive description of the system
- it makes explicit provisions for adding new types of components
- It can express and check appropriate compatibility restrictions and configuration constraints.
- It accepts existing code as components, incurring no runtime overhead after initialization.
- It allows easy incorporation of specifications and associated analysis tools developed elsewhere.

The implementation provides a base for extending the notation and validating the model.

2.1.3. Ajax

Ajax [24] is designed to help programmers understand large Java programs by capturing global invariants using static program analysis. Ajax provides a set of tools, each addressing a specific programmer task, and a common, reusable static analysis infrastructure used as the basis for the tools.

Ajax makes contributions in the following areas:

- Ajax provides a tool to statically check the usage of Java *downcast* operations. In real Java programs, including programs obtained from Sun and NASA, Ajax is able to prove that more than fifty percent of the downcast operations are safe and will never fail at run-time. This information can be used by compilers to speed up execution, and by programmers to eliminate the possibility of bugs associated with these downcasts. Ajax is currently the only system with this capability.
- Ajax provides a tool to compute object models of Java programs. Using advanced static analysis, it can compute more refined and detailed models than competing tools, revealing more information to the programmer.
- Ajax provides several other tools to statically compute call graphs, find dead code, and scan programs for accesses to particular data objects.
- All these tools use a common, reusable analysis infrastructure. One of the key contributions of Ajax is its clean separation between static analysis engines and the different tools that consume analysis results. The interface is based on a simple, formally defined static abstraction of program behavior called the *value-point relation* (VPR). The VPR captures generalized aliasing behavior of a program.
- Multiple implementations of the analysis interface are possible. Ajax provides two basic analysis engines: a fast, simple engine that exploits declared Java type information, and a more sophisticated engine called SEMI[1] based on polymorphic type inference. Analysis engines can also be combined to give "the best of both"; our results show that this often provides significantly better results than either alone.
- SEMI is based on type inference with polymorphic recursion, which provides cheap, robust context-sensitive analysis. SEMI is the only static analysis engine for Java that provides polymorphic recursion. The SEMI engine also solves some difficult (and hitherto unaddressed) scalability issues in combining polymorphic recursion with analysis of records containing multiple distinct fields.
- Ajax tools can be configured with different analysis engines depending on the task at hand. At runtime, the system can select a cheap, crude analysis or a sophisticated, expensive analysis. Our results show that for some tools, a cheap analysis is as good as any, whereas other tools benefit significantly from more expensive analysis.

---

[1] SEMI is not an acronym – it is an abbreviation of the term *semiunification*.

All these contributions are developed in the context of analyzing the full Java language. Ajax analyzes the JDK class libraries as well as real applications such as the "javac" Java compiler, the "Jess" expert system shell, the "javafig" graphical diagram editor, and many others.

2.1.4. Composable Connectors

In many situations, specialized forms of interaction are needed to bridge component mismatches or to achieve extra-functional properties (e.g., security, performance or dependability), making the design and implementation of these interaction mechanisms a critical issue. But creating new connectors *ab initio* is difficult and time-consuming. The idea of composable connectors arises from this need for a more principled and systematic way to understand, describe, and generate new kinds of connectors.

We have described a set of operators, or "connector transformations", that can augment generic communication mechanisms (such as RPC and publish-subscribe) to incrementally add new capabilities to an existing connector. These transformations are generic, with the benefit of yielding a smaller overall number of transformations to understand: for example, we identify a generic transformation "data transform" that applies a function (perhaps for compression or error correction) to the data communicated on the connector, rather than identifying domain-specific instances such as "add compression" and "add error correction codes" as transformations.

These simple transformations are compositional. Complex and domain-specific results can be obtained by applying sequences of general-purpose transformations; we have identified some domain-specific patterns of usage (such as a sequence that can be used to add Kerberos authentication). Thus a new connector can be described as a base connector plus a sequence of specific transformations that together produce the intended modification (such as, authentication). Such a description can be informal text with reference to patterns of usage; we also use Wright as a basis for more formal descriptions of the individual connector transformations.

Initial work on a tool to generate the implementation of new composed connectors indicates that the compositional approach will allow rapid, easy development of complex connectors that are good enough for many practical systems; the compositional approach can provide a middle ground between the time and expense of connectors hand-tailored to the system requirements, and the selection of off-the-shelf connectors that may not provide a good fit to the requirements.

We have built a tool to perform composable connector transformations. It can be used to produce implementations of a variety of complex connectors by modifying an existing simpler connector, with less effort than would have been required using a more traditional approach. The tool is not specific to a particular domain; examples of use include security-related modifications (e.g. adding Kerberos authentication) as well as dependability-related (e.g. adding error detection and retry).

The current implementation of our tool operates on Java Remote Method Invocation (RMI). When no transformations are used, the tool naively generates a Java RMI

connector; this is the base connector (the tool has knowledge of what initialization steps to add, etc., to prepare for and perform a remote method invocation).

The user of this tool must first break down the complex modification (e.g. "adding authentication") into a sequence of simpler connector transformations that together achieve the intended result; these transformations will be performed in order by the tool. The user of the tool must also supply, for each desired transformation, code fragments. In addition to the kind of transformation desired and its associated code fragments, the tool is also given as input the location of source files for the components that are to communicate. Given these inputs (type of transformation, code fragments, existing source files) the tool generates a new connector (producing modified source files, and composable wrappers created from the remote object's interface). Transformations can be composed.

Each transformation inserts its code fragments at locations that are specific to that transformation; this may be at particular sites in the input source files, or the fragments may appear in newly generated composable wrappers. (Here the division of labor between the tool and the user is apparent; the tool undertakes to know the appropriate points in the base connector at which, for example, initialization steps may occur, but the tool itself has no domain-specific knowledge of, for example, appropriate security library calls; this is supplied by the user of the tool in the code fragments, which would chiefly include the calls to those pertinent library methods.)

For most transformations the transformed connector (including generated wrappers) presents essentially the same interface to the connected components as the original connector did, minimizing the impact on the component implementations. Transformations that change the connector's interface are also possible and are desirable in some cases; for example, to produce a connector that overcomes a mismatch between the connected components.

A modification to a connector is generally non-localized and requires changes in multiple code (and even non-code) artifacts. By performing the widely scattered insertions of modification code fragments, the tool helps to ensure that the modification is carried out consistently, in addition to saving some time and effort. The Acme architecture description language provides a convenient way to collect the fragments and transformation directives (as well as pointers to component implementations) in a single location as properties of a connector type, using a system's architectural description to drive the generation/transformation of its connector implementations; future work will enable an Acme description to actually drive the connector transformation tool.

2.1.5. Acme integration with Rapide tools

Raparch is a graphical framework to

- address various architectural issues in one framework,
- facilitate an evolutionary architecture development, and
- preserve roles of an architecture in system evolution.

Raparch constructs an architecture based upon the concept of an interface connection architecture (ICA). The ICA, in essence, raises the level of visibility of interactions among components by specifying both provided and required features at the interface of a component. With such an abstract communication infrastructure an architecture specification can be separated from system implementation. It enables us to construct an executable architecture even before a system is built.

Raparch defines four formal models of such an architecture based upon levels of abstraction: conceptual, structural, behavioral and execution models of an architecture.

The Raparch framework provides us with the features for an evolutionary architecture development. It

- provides explicit graphical notations for architectural elements,
- maintains a consistency between levels of abstraction of architecture,
- makes architecture evolution tractable and enables us to specify both static and dynamic architectures,
- promotes reuse of architectural patterns for commonality,
- manages and guides an architecture's evolution throughout the system's life cycle,
- enables us to reason about the system based on simulation results, and
- boosts confidence in regard to correctness of system implementation.

The original Raparch was developed as a graphical front-end for specifying a Rapide architecture [20]. The current Raparch has been extended to become a graphical framework for developing an interchangeable architecture specification in other ADLs. Raparch attempts to consolidate and visualize an entire architecting process in one framework. The framework turns an architecting process into an integral circle of design iterations from architecture specification to validation based upon four formal models of an architecture. It provides a closure in iterative and incremental architecting process by

- reinforcing relations between graphical notations and the underlying formal semantics of models of an architecture, and
- integrating tool suites in the framework.

Raparch currently includes tool sets for the Rapide ADL such as a compiler (rpdc), a poset browser (pov) and an animator (raptor). Source code for Rapide architectures can be synthesized and the architecture simulated within the Raparch framework. The evolutionary architecture process has been successfully demonstrated in transaction processing system architecture examples. The underlying infrastructure in Raparch should be able to specify an architecture not only in Rapide ADL but also in other architecting methodologies such as Acme or UML. The work is in progress to complete specifications in Acme and UML ADLs for the same architecture.

ACME in Raparch:

Acme focuses on architectural structures of systems and leaves the semantics for an architecture to interpretation by specific ADLs. Acme provides seven constructs for architecture descriptions: components, connectors, systems, ports, roles, representations

and representation maps. In addition Acme provides a mechanism for the annotation of architectural structure with property lists.

An Acme architecture description of a system can be synthesized in a relatively straightforward manner within the Raparch framework, because architectural elements in Raparch have strong correspondences to Acme constructs.

An Acme component is well represented by a component interface in Raparch. The features of a component interface in Raparch are mapped into ports in Acme. The interface types of components in Raparch correspond to types in Acme. The hierarchical relations of architectural elements in Raparch can easily be translated into representations in Acme. The components' behaviors are encoded in the form of annotations in property lists in Acme. The architectural configuration in Raparch is easily transformed to a system description in Acme.

The correspondence between a Raparch connection and an Acme connector is a subtle one. For a simple connection, it is straightforward: interacting features of a component interface in Raparch are well mapped into roles of a connector in Acme. Both uni- and bi-directional connections (in Raparch) and connectors (in Acme) are supported.

A complex communication protocol in Raparch is usually embodied in interacting components' behavior with connections. For an Acme connector for a complex protocol, Raparch must convert a complex protocol into a specific communication component and simple connections.

UML in Raparch:

UML is a family of graphical notations to describe the attributes of and relations between architectural elements. UML provides nine diagrams for a easy-to-communicate, understandable, and maintainable architecture description for a system. It is excellent for the documentation of architecture development, but unfortunately does not provide any executabilities or analytic capabilities for an architecture.

The ideas of UML diagrams can be well captured in models at different levels of abstraction of an architecture in Raparch:

- A conceptual model in Raparch is mapped onto a use-case diagram;
- The type interfaces of components and their relations are represented in a class diagram;
- The reactive rules of an individual component in a behavior model are translated to a state diagram;
- The connection rules with behavior of interacting components are transformed to a sequence and/or collaboration diagram;
- An object diagram is well represented by a structural model in Raparch. A flow of activities in an activity diagram is well represented by an execution model of an architecture;

A component diagram and a deployment diagram are not supported in Raparch. However, the hierarchy of an architecture of a system in Raparch is well captured in a package diagram in UML

2.2. Design analysis

2.2.1. Wright

A stumbling block to architectural design is the inability to characterize software architectures precisely and to reason about their properties. In this project we attacked this problem by adapting existing formalisms so that they can be used by practitioners to describe and analyze complex software architectures. The adaptations allow the formal notations to:

- better match the descriptive needs of software architects,
- provide analysis and consistency checking as built-in features (rather than having to be constructed from scratch for each new specification), and
- scale to real applications.

One main focus of that work in this project was the development of a formal architecture description language, called *Wright*. This language has two novel features. First, it permits the description of *connector types* as first class entities. A connector is specified (in a variant of CSP) as a set of interactions between two or more components. The formal notation supports analysis and automated checking of properties such as the consistency and completeness of architectural descriptions with respect to communication behavior. In particular, the analyses can pinpoint mismatched behavior of interacting components, identifying errors that would cause those components to function incorrectly in the running system.

Second, like Acme, Wright allows one to describe *architectural styles*. A style is a family of systems that share a common vocabulary of component and connector types, and that satisfy certain constraints on topology and behavior. Common commercial examples include Visual Basic, CORBA, and Unix Pipes, although there are also numerous domain-specific styles such as the HLA (described below). By characterizing architectural styles, Wright allows architects to be explicit about recurring structural idioms and system organizations. Further, Wright allows us to prove general theorems about a style – theorems that then apply to every instance of the style. For example, for many styles it is possible to prove a general "closure" theorem, which shows that architectural subsystems can be treated as a primitive component. (Such results are useful because any theorem that is true of a style's primitive component types can be extended to certain encapsulated subsystems in that style.)

Wright formed the basis of Robert Allen's PhD thesis [1], and an article on it appeared in the July 1997 *ACM Transactions on Software Engineering and Methodology* (TOSEM) [2]. We have applied Wright to several significant case studies. We used it to analyze a published DoD standard for distributed simulation, called the High Level Architecture (HLA) for Distributed Simulation (http://www.dmso.mil/projects/hla/) as

detailed in Section 4.1. The flaws that we detected in their published standard (over 80 of them) have drawn considerable attention, and resulted in our helping the Defense Modeling and Simulation Office draft a revised standard – including our authoring a revision of one of the more complex parts of the standard. Additionally, some of our formal models are currently included as part of the revised standard. We also have used it to specify the Enterprise JavaBeans framework and part of the JavaPhone standard, described in Sections 4.2 and 4.3.

To illustrate the use of Wright, a simple Client-Server system description is shown in Figure 4. This example shows three basic elements of a Wright system description: component and connector type declaration, instance declarations, and attachments. The instance declarations and attachments together define a particular system configuration.

```
Configuration SimpleExample
  Component Server
    Port Provide <provide protocol>
    Computation <Server specification>
  Component Client
    Port Request <request protocol>
    Computation <Client specification>
  Connector C-S-connector
    Role Client <client protocol>
    Role Server <server protocol>
    Glue <glue protocol>
  Instances
    s: Server
    c: Client
    cs: C-S-connector
  Attachments
    s.Provide as cs.Server;
    c.Request as cs.Client
end SimpleExample.
```

Figure 4. A simple client-server system specified in Wright.

In Wright, the description of a component has two important parts, the *interface* and the *computation*. A component interface consists of a number of *ports*. Each port defines a point of interaction through which the component may interact with its environment.

A connector represents an interaction among a collection of components. For example, a pipe represents a sequential flow of data between two filters. A Wright description of a connector consists of a set of *roles* and the *glue*. Each role defines the allowable behavior of one participant in the interaction. A pipe has two roles, the source of data and the recipient. The glue defines how the roles will interact with each other.

Each part of a Wright description – port, role, computation, and glue – is defined using a variant of CSP [15]. Each such specification defines a pattern of events (called a process) using operators for sequencing ("→" and ";"), choice ("Π" and " □ "), and parallel composition ("||").

Wright extends CSP in three minor syntactic ways. First, it distinguishes between *initiating* an event and *observing* an event. An event that is initiated by a process is written with an overbar. Second, it uses the symbol § to denote the successfully-terminating process. (In CSP this is usually written "SKIP".) Third, Wright uses a

quantification operator: <*op*> $x : S \bullet P(x)$. This operator constructs a new process based on the process expression $P(s)$, and the set S, combining its parts by the operator <*op*>. For example, $[]\, i:\{1,2,3\} \bullet P_i = P_1 \,[]\, P_2 \,[]\, P_3$: i.e., a choice among one of three processes, P_1, P_2 or P_3. Similarly, $; x:S \bullet P(x)$, is a process that consists of some unspecified sequencing of the processes:

$; x:S \bullet P(x) = \Pi\, x:S \bullet (P(x) \; ; (; x:S \setminus \{x\} \bullet P(y)\,)\,)$.

For example, a simple client role might be defined by the CSP process:

```
Role Client = (request → result?x → Client) ⊓ §
```

This specification defines a participant in an interaction that repeatedly makes a request and receives a result, or chooses to terminate successfully.

2.2.2. Ladybug

Ladybug [6] is an automatic design checker. The user specifies the abstract state of a system's design in terms of a set of state variables and finite domains of values over which the variables range. These domains are expressed in terms of binary relations, functions, and sets. A model is an assignment of variables to values. The user also writes a first-order formula over state variables, which is the specification against which the design is checked. Because all domains are finite, we could in principle exhaustively check the formula against every single model of the system. Ladybug, however, uses a combination of techniques, collectively called "selective enumeration," that drastically reduces the number of models to check. These techniques include novel ones such as bounded generation, tried and true ones such as derived variables, and those used in other domains such as isomorph elimination. The strength of Ladybug is in combining these techniques into one system, letting the user apply each individually or together.

Ladybug has been applied to a suite of benchmark examples. For some of these benchmarks, without these techniques what Ladybug can do in minutes would otherwise essentially take forever. The most non-trivial case study was to apply Ladybug to the HLA protocol standard (of the distributed simulation community). Through the careful writing of formal models and specifications and the use of checking, we found numerous flaws in the protocol.

2.2.3. Revere

Revere [18] is an automatic verification tool. It is based on a novel technique called theory generation. While we have applied it to reasoning about authentication protocols, the technique is not specific to authentication. A formal system (for us, a subset of first-order logic) is expressed as a set of axioms and rules. A protocol is expressed in the same language of the logic as additional rules. The theory generation algorithm then applies all axioms and rules, deriving new formulas that are in the theory of the protocol. We guarantee that the algorithm halts because of conditions on the family of logics; these conditions are for practical examples syntactically checkable. Once we have a finite representation of the theory of the protocol, verifying the protocol for properties is as simple as a membership check: is this formula in the theory? Thus theory generation is a simple verification technique.

We built Revere so that it is parameterized over a given logic. Thus it produces a logic-specific theory generator. We can then use the specific theory generator to verify properties of protocols. We applied Revere to a set of authentication logics, including the Burrows-Abadi-Needham Logic of Authentication [5] to reason about the classic suite of authentication protocols. We revealed known flaws and discovered new ones. We also made explicit many implicit assumptions in either the logic or the protocol design. In one case, we found an optimization of the protocol. We have also applied Revere to Kailar's accountability logic [17] and verified a simple e-commerce protocol.

In all our case studies, the time it takes to generate theories is on the order of a few seconds; the size of the finite representations of the theories is on the order of tens of formulas (easily under 100). So this verification technique is fast, simple, and completely automatic.

3. Enabling correct modifications

The techniques and technologies discussed above focused on describing and analyzing software systems (both at an architectural and implementation level), and are effective for specifying the structure and properties of a software system at a particular point in time. In aggregate, this work goes a long way towards enabling the design of the system to be recorded, and allowing analysis of the design to verify its correctness. To support software evolution, however, it is necessary to provide tools and techniques for capturing and analyzing software architectures as they evolve over time. We have investigated three approaches to accomplishing this, which are elaborated in subsequent sections. The first is to explore extensions to Acme that permit a designer to include the specification of design constraints that on a system architecture. Such a technique provides the designer with a more powerful mechanism for specifying the vocabulary of a design and describing how the design should evolve over time.

Another mechanism that we have begun to explore is the use of maps, which allow a designer to capture the correspondence between different views of an architecture, or indeed different architectures. Among other things, maps can be used to record the meaning of a change between different versions of a design as it evolves over time.

The final mechanism we have explored has been the correspondence between an architecture and its run-time implementation. We do this by being able to associate an activity with an architecture. An activity represents a set of runtime events, specified in an architectural context, that occur as a program runs. We have specified an extension to Acme to include activities, and future work involves building tool support to include the generation of these events and their interpretation by architectural tools.

3.1. Design constraints

We have developed a language called Armani [22], [23], an extension to Acme that allows the specification of design constraints. Specifically, it allows an architecture to be annotated with constraints that bound the way that a system's architecture, or the properties of that architecture, can evolve over time. These constraints can be used by subsequent developers to help maintain the system's conceptual integrity as the system is updated and evolved. Useful architectural constraints include: limitations on the

modifications to a communications topology; restrictions on legal topological patterns of a system's structure graph; maintaining performance, reliability, security, fault-tolerance, and other parameters within acceptable ranges; and ensuring interface compatibility.

Armani uses a first order predicate logic (FOPL) formalism as the basis for the constraint specification language. Armani addresses the problem of the undecidability of FOPL by ensuring that predicate quantification is done only over finite sets. In addition to FOPL primitives, Armani defines a set of primitive predicates that are architecture-specific. In all, Armani defines twenty-four primitive functions, divided into four categories. The four categories are:

Topological These functions allow an architect to specify constraints such as how and with what a system's components can communicate, which components must be connected, which must not be connected, etc. These functions can be used to enforce parent-child relationships between entities in a design.

Properties These operations access properties and substructures of design elements. The operator "." is used to identify specific element properties or children of design elements.

Aggregation These functions allow sets of substructure to be referenced by name. For example, p.Roles names the set of rolls associated with a connector p.

Types These functions can be used to determine or select design elements based on the elements type declarations.

Examples of the primitives from the four categories are presented in Table 1. All design constraints that may be specified are not equally important. Some may be guidelines about how or whether aspects of a system can be changed. Some constraints should never be violated because they contain key design principles and assumptions; violating these constraints could render the system unusable. Armani caters for these two levels of constraints by allowing designers to specify whether a constraint is *heuristic* or *invariant*. Heuristic constraints are intended to specify rules of thumb, and can therefore be violated. Invariant constraints, however, specify constructs that must be maintained by the design at all times. Together, heuristic and invariant constraints further allow the evolution of a design to specified and guided.

Function category and signature	Function description
Topological: Connected(comp1, comp2)	Returns true if component comp1 is connected to component comp2 by at least one connector, otherwise it returns false.
Topological Reachable(comp1, comp2)	Returns true if component comp2 is in the transitive closure of Connected(comp1,*), otherwise it returns false.
Properties: HasProperty(x, propertyName)	Returns true if element x has a property called propertyName, and false otherwise.

Properties: <ElementName>.<PropertyName>	Returns the value of the property identified by <PropertyName> in the element <ElementName>.
Aggregation: <SystemName>.Connectors	Returns a set containing all the connector instances in the system identified by <SystemName>.
Aggregation: <SystemName>.Components	Returns a set containing all the component instances in the system identified by <SystemName>.
Types: DeclaresType(elt, typeName)	Returns true if the element identified by elt declares the type identified by typeName, and false otherwise.

Table 1. An example of the primitive architectural functions in Armani.

We have also developed tools to support Armani constraint checking, including integration with AcmeStudio and Visio tools. AcmeStudio allows constraints to be defined as properties to elements, and the Armani constraint-checking tool may be invoked to provide feedback to the designer in AcmeStudio about the validity of their design. The full specification of Armani, and its relation to Acme, can be found in [22].

3.2. Resolving architectural mismatch

As discussed in [11], some problems of composition are due to low-level issues of interoperability, such as mismatches in programming languages or database schemas. However a more pervasive class of problem is *architectural mismatch*. This problem relates to assumptions that are made about the environment and protocols used in software components. Such assumptions include those made about the nature of components and connectors (in terms of construction, protocols or data models, for example). Architectural mismatch is especially troublesome because these assumptions are not documented, and so are usually made apparent when an attempt is made to integrate components. We have investigated two methods of addressing architectural mismatch. The first is Flexible Packaging; the second is documenting the relationship between architectures using maps.

3.2.1. Resolving architectural mismatch with flexible packaging

To integrate a software component into a system, it must interact properly with the system's other components. Unfortunately, the decisions about how a component is to interact with other components are typically committed long before the moment of integration and are difficult to change. We developed the Flexible Packaging method [7], [8], which allows a component developer to defer some decisions about component interaction until system integration time. The method divides the component's source into two pieces: the ware, which encapsulates the component's functionality; and the packager, which encapsulates the details of interaction. Both the ware and the packager are independently reusable. A ware, as a reusable part, allows a given piece of functionality to be employed in systems in different architectural styles. A packager, as a reusable part, encapsulates conformance to a component standard, like an ActiveX

control or an ODBC database accessor. Because the packager's source code is often formulaic, a tool is provided to generate the packager's source from a high-level description of the intended interaction, a description written in the architectural description language UniCon. The method and tools are evaluated with two case studies, an image viewer and a database updater.

3.2.2. Managing the correspondence between architectures with maps

In practice, while describing software architecture precisely is an important step towards making architecture design more rigorous, it often helps to have multiple views of the same system. For example, Kruchten proposes a model in which five concurrent views of the same software architecture capture different set of concerns [19]. The Meta-H notation requires a hardware view that shows the hardware structure and a software view that shows the software application running on the hardware [4]. Multiple views could also be the result of system evolution, in which case different versions of the system may be represented using different views.

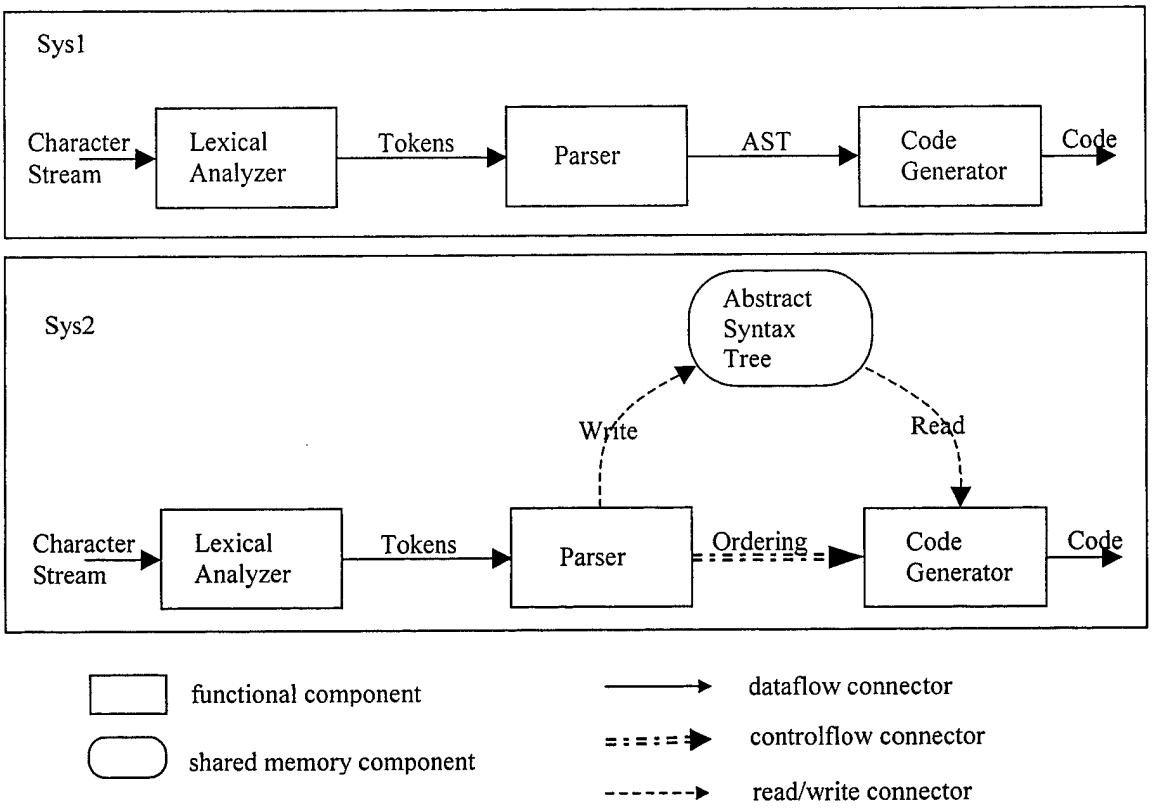


Figure 5. Two architectures for a compiler.

In all of these cases, there are important relationships between these views. Those relationships might show, for example, which part of a software architecture runs on which processor or which part of one architectural design has the same functionality as that of an alternative design. Knowledge of those relationships helps designers to better understand the system and to reason about it.

To illustrate our approach, consider two views of a design of the architecture for a compiler. An abstract pipe-and-filter architecture is depicted at the top of Figure 5 as Sys1. A more concrete (i.e., implementation-oriented) architecture that is a hybrid of pipe-and-filter and shared-memory styles is shown at the bottom of Figure 5 as Sys2.

```
map compMap with {
   Sys1.analyzer to sys2.analyzer;
   Sys1.parser to sys2.parser;
   Sys1.generator to sys2.generator;
   invariant Sys1.analyzer.throughput == Sys2.analyzer.throughput;
   invariant Sys1.parser.throughput == Sys2.parser.throughput;
   invariant Sys1.generator.throughput == Sys2.generator.throughput;
}

map pipe2Channel with {
   Sys1.ast to
      [ from = Sys2.write;
        through = Sys2.ast;
        dest = Sys2.read;
        control = Sys2.ordering
      ];
}
```

Figure 6. An example of two maps in the Acme mapping extension.

We have developed a proposed extension to Acme and have provided support for this extension in the AcmeLib tool library. Figure 6 provides an example of the Acme description of the system in Figure 5. The first map, compMap, describes the correspondence between three components in the each architecture. This can be read as equating, for example, the analyzer component in Sys1 to the analyzer component in Sys2. Additionally there are three Armani invariants associated with this map that specify that the throughput of each associated component in the map must be the same.

The first map is a simple map where the correspondence between components is one-to-one. However, this is unlikely to be the case in widely different views of an architecture. The second map, pipe2Channel, illustrates a mapping between one connector in Sys1 to four connectors in Sys2. The example uses tuples to distinguish various roles that each connector plays in the map.

The mapping extension is still in a prototyping state, and the specification as been disseminated to other EDCS participants for comment. For a full description of the mapping extension, including other types of aggregation and typing, refer to [15].

3.3. Correspondence between an architecture and its implementation

Many architectural descriptions have ways of describing and analyzing architectural behavior in terms of the architecturally significant *events* that can be exhibited at run time. We call a collection of such events an *activity*. For example, Wright describes behavior in terms of event patterns defined in a subset of CSP [2]. Rapide also describes behavior in terms of event patterns. In addition, Rapide and several tools developed by others allow one to monitor architectural behavior and then analyze the results, or "play it back" as a form of architectural animation [20].

Given this area of commonality and the general need for event-based architectural analysis, a next obvious step is to find ways to represent events in such a way that different event-based architectural tools can work together. Ideally, this would lead to common models for representing and analyzing event behavior.

A first step towards sharing event-based behavior is to find a standard way to represent events and collections of events. With an eye toward this goal, a number of researchers at ISI (Wile), CMU (Garlan & Kompanek) and Stanford (Luckham & Kenney) started a dialogue to converge on an event standard. The results are summarized in this working report. The main idea of the proposal is to adopt an Acme-like approach: a simple base-level event representation would capture the minimal, core aspects of events. Additionally, other more tool-specific information could be added to those event representations in the form of annotations.

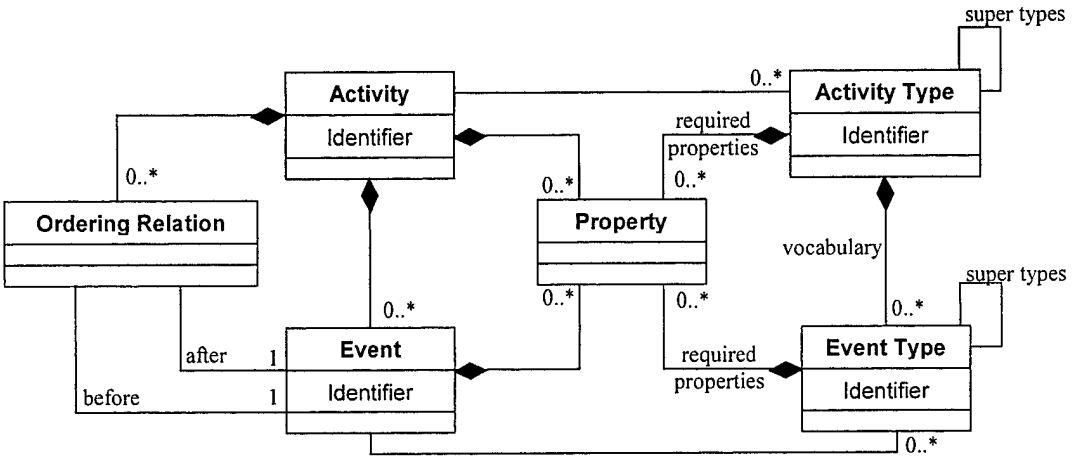


Figure 7. Activity language concepts.

Figure 7 illustrates the UML diagram indicating the main conceptual elements of the proposed activity language and their relationships. The concepts are described below:

Activity — An activity is a description of a behavior consisting of a set of *events*, a set of *ordering relations* over those events, and a set of *properties* that describe auxiliary information associated with the activity. An activity may also be associated with zero or more *activity types* (see below).

Event — An event is a uniquely identified behavioral occurrence that takes place within some system. An event has a set of properties that describe the nature of the event. An event may be associated with zero or more *event types* (see below). An event could have one or more representations, each representation being an activity.

Property — A property is a typed attribute-value pair used to encode semantic information about an event or activity.

Ordering Relation A set of ordering relations is used to define the temporal (or causal) structure of events within an activity. In general, an ordering relation will determine a partially ordered set (poset) in which certain events are related to others. In posets, event order is described by a set of explicit ordering relationships which specify pairwise ordering relationships between events in an activity. An important special case of a poset is a sequence (often called a trace).

Activity Type An activity type defines a family of activities. The members of this family are determined by a predicate associated with that type, which defines a set of constraints that each activity (instance) must satisfy. Constraints include such things as required properties of the activity, the event types that can be included in an activity (i.e., the event vocabulary), and a specification of how events are ordered within an activity of this type. This proposal does not provide a means to declare such an ordering pattern, and it is not its intention to provide one. Given that there are so many notations for declaring the pattern of events (state charts, regular expressions, CSP, just to name a few), and various communities accept one or more of these notations, we decided not to choose any one as our "standard" notation. Instead, any notation could be used to specify an ordering pattern as a property of the activity type. An activity type may also describe other constraints that must be satisfied by an activity (not yet included in this proposal). An activity type can be defined as an extension/subtype of an existing set of activity types, meaning that it satisfies the constraints of all of those types, in addition to any others that it defines.

Event Type An event type defines a family (or vocabulary) of events. The members of the family are determined by a predicate associated with that type, which defines a set of constraints that each event (instance) must satisfy. Constraints include such things as a set of properties that all instances of this type must have, relationships between those properties, etc. An event type may be defined by extending an existing set of event types.

An activity language extension to Acme based on these properties has been proposed and added to the AcmeLib set of tools. An introduction to this is beyond the scope of this report. The full activity language proposal can be found in [13]

4. Evaluation

In this grant, we have claimed to reduce the cost of evolving software by being able to specify and analyze architectural designs, and to be able to support better modification to these designs. The technologies discussed have been applied to a number of industry standards with very encouraging results. The following sections summarize the use of architectural modeling and analysis on the Defense Modeling and Simulation Office's

High Level Architecture integration standard, and Sun Microsystems Enterprise JavaBeans and JavaPhone standards. We also briefly discuss our efforts to clarify the relationship between architectural design and object-oriented design as represented by the Unified Modeling Language (UML).

4.1. High Level Architecture (HLA)

An increasingly important trend in the engineering of complex systems is the design of component integration standards. Such standards define rules of interaction and shared communication infrastructure that permit composition of systems out of independently-developed parts. A problem with these standards is that it is often difficult to understand exactly what they require and provide, and to analyze them in order to understand their deeper properties.

In the research carried out in this project we demonstrated how one can use formal architectural modeling to clarify these kinds of issues. The key idea is to treat the integration standard as a structured protocol, using the Wright ADL The description can be analyzed using formalisms and tools for modeling software architecture. By making explicit the protocol inherent in the integration standard, we are able to make precise the requirements on both the components and on the supporting infrastructure itself. This in turn provides a deeper understanding of the standard, and supports analysis of its properties.

There were a number of technical hurdles that make this research non-trivial. First, most component integration standards are relatively complex, often involving dozens of routines in their API. Structuring becomes a central issue for modeling. Second, for a complex standard it is critical that the formal model be traceable back to the original documentation. This is because when errors are found, it must be possible to relate the results back to the source. Third, is the issue of variability in the standard. It is critical to distinguish between aspects of the model that are fixed by the standard and those that are allowed to vary from one system to another. In practice this can be difficult to do because a particular API may make implementation choices that are not intrinsically part of the integration standard. Fourth is the problem of tractability. If the formal model is to be useful to humans or to analysis tools it must be simple enough that it can be understood (or mechanically processed), but detailed enough that useful properties are revealed

The result of this research was the detection of a number of flaws in the published HLA standard (over 80 of them). This work has drawn considerable attention, and resulted in our helping the Defense Modeling and Simulation Office draft a revised standard – including our authoring a revision of one of the more complex parts of the standard.

To illustrate the kinds of ambiguities we uncovered, here are two examples:

- **Exceptions**: Each service description in the HLA specification lists a set of exceptions. In our attempt to formalize the HLA, we realized that the formalization (and presumably any implementation) wasn't possible unless we knew if these exceptions resulted in actual message traffic or whether they were simply anomalies that should be considered (but without explicit

notification). It turned out that the answer was that in some cases exceptions are used to convey important information, while in other cases they represent genuine errors.

- **Retained state**: To mediate the communication between federates, the RTI must retain certain state. But it is not clear what state, and for how long. For example, when a federate saves its state, it provides a save label. State can be restored through a "restore" service call (using an existing label). But state can only be restored when all federates have a save for the save label being restored. However, in the HLA specification there is no indication of how long this save label can be successfully used: after what point can a federate discard a previous save?

In addition to raising critical issues for clarification, the formal model also helped expose unintended behavior of the standard. We discovered about a dozen such anomalies using a combination of careful review and the facilities of a commercial model checker for CSP, called Failures Divergence Refinement (FDR) [10]. To make use of the model checker we used two primary techniques. The first was to look for potential deadlocks in parts of the specification.[2] When the tool detects "deadlock" it provides a trace showing where the process goes awry. Such deadlocks typically indicated the presence of a situation in which different parts of the specification had inconsistent views about the behavior expected of other parts. The other technique was to see if the model was consistent with some desirable behavior. To check for this situation we used to tool to check if a refinement relationship exists between the model and a process that exhibits just that behavior.

The problems that we detected fell into three classes:

- **Race conditions:** Situations where unexpected behavior could occur if the orderings of distributed interactions were allowed to vary as originally specified.
- **Deadlocks:** Situations where the simulations could "get stuck" waiting forever for parts of the system to respond.
- **Unexpected outcomes:** The Wright model allowed us to analyze whether certain combinations of behaviors could lead to unintuitive outcomes.

4.2. Architectural analysis of Enterprise JavaBeans™

Another example of component integration standards is Sun's Enterprise JavaBeans™ (EJB) architecture [9]. EJB is intended to support distributed, Java-based, enterprise-level applications, such as business information management systems. It prescribes an architecture that defines a standard, vendor-neutral interface to information services including transactions, persistence, and security. EJB thereby permits application writers to develop component-based implementations of business processing software that are portable across different implementations of those underlying services.

---

[2] In principle one could run the entire model through FDR and find all deadlocks within. In practice, the HLA model is much too large for the checker: so we had to break it into small pieces, and incrementally recombine these in various combinations.

One critical issue for users and implementers of a component integration standard is the documentation that explains what the standard provides and what is required to instantiate it correctly for some application. Typically, component integration standards are specified using a combination of informal and semi-formal documentation. On the informal side are guidelines and high-level descriptions of usage scenarios, tips, and examples. On the semi-formal side one usually finds a description of an application programmers' interface (API) that explains what kinds of services are provided by the standard. Although voluminous, documentation such as this has two intrinsic problems. First, related information is spread throughout the document. For example, to determine what sequence of method calls a bean must follow to request a typical service from the container, the reader must locate the explanation in the text (hopefully covering all relevant operations), refer to the API method descriptions, examine any examples of sample executions, and consult the list of possible raised exceptions. Second, the lack of a precise definition makes it difficult for a reader to resolve inconsistencies and ambiguities, and to determine the intended semantics of the framework. As an example of irresolvable inconsistencies, in one place the documentation states the Home Interface should "define *zero* or more *create* methods" (page 14), while in another it specifies "*one* or more *create* methods" (page 20). Without a single place in the document that has the precise definition, it is impossible to determine which of the two (if either) is correct (even assuming we can determine what exactly a *create* method should do).

The work we did on EJB illustrates how one can use formal architectural modeling to provide an abstract structural description of component integration standards. We built a formal model of EJB using Wright, which focuses on structure and protocols of interaction, highlighting potential deadlocks and possible composite behaviors of intervening parts. Wright, however, does not handle issues such as performance, reliability, or security.

The architectural model we built using Wright makes clear the high-level interfaces and interactions among principal parts, and characterizes their semantics in terms of protocols. By making explicit the protocols inherent in the integration framework, we make precise the requirements on both the components and on the supporting infrastructure itself. By precisely specifying the implied protocols of interaction for EJB, we achieve a number of immediate benefits. First, the formal model is explicit about permitted orderings of method calls, and about where the locus of initiative and responsibility lies. Second, the model makes explicit where different parts of the framework share assumptions, and therefore the dependencies between parts. Third, the model helps clarify some of the more complex aspects of the model by exposing all possible behaviors. Analysis using FDR revealed one significant problem concerning a possible race condition between the delegation and swapping (*passivation*, in EJB terminology) processes inside the EJB *Container* part. While arguably one might attribute the detected problem to *our* specification, and not to Sun's EJB standard, it does point out a place where the complexity of the standard can lead to errors that are hard to detect. Without a precise model and effective automated analysis tools to identify problem areas, errors are easily introduced, undetected, into an implementation.

The primary contributions of this work are twofold. First, we showed how formal architectural models based on protocols can clarify the intent of a component integration

standard, as well as expose its critical properties. Second, we describe techniques to create the model, and structure it to support traceability, tractability, and automated analysis for checking of desirable properties. These techniques, while illustrated in terms of EJB, shed light more generally on ways to provide formal architectural models of object-oriented frameworks, thus contributing to the understandability and maintainability of those frameworks. Our work on EJB is described in more detail in [25].

4.3. Architectural analysis of Enterprise JavaBeans

This work provides a brief guide to architectural documentation, focusing on the reality of industrial development, most notably on the use of the UML object modeling language as a support to architectural representation. The work stems from the increasing recognition that well-designed software architecture is critical to the success of any complex software-related project. By exposing the key system design concerns, a properly designed architecture goes a long way towards guaranteeing that a system will satisfy its principal requirements and helps insure system integrity as the system evolves over time. By providing an abstract description of a system, the architecture exposes certain properties, while hiding others. Ideally, an architectural representation provides an intellectually tractable guide to the overall system, permits designers to reason about the ability of a system to satisfy certain requirements, and suggests a blueprint for system construction and composition. Despite the variability that is caused by the specificity of each domain, the software architecture community seems to be converging on four classes of views for architectural documents:

Context-based views: indicate the setting in which the system is to be employed, and often identify the abstract domain elements that determine the system's overall requirements and business context.

Code-based views: describe the structure of the code, indicating how the system is built out of implementation artifacts, such as modules, tables, classes, etc. Such views are particularly useful as a guide to implementation and maintenance. Special, but common, cases of code-based views are layered diagrams and class diagrams.

Run-time views: describe the structure of the system in operation, indicating what are the main run-time entities and how they communicate between each other. Run-time views allow one to reason about behavioral properties as well as about attributes such as run-time resource consumption, performance, throughput, latencies, reliability, etc.

Hardware-based views: describe the physical setting in which the system is to run, indicating the number and kinds of processors and communication links. The information contained in these views is often combined with that in run-time views to derive system performance properties.

We examined Sun Microsystem's JavaPhone™ [26] as a case study to illustrate the approach: what kinds of information are provided in each kind of view, what forms of notation should be used, how these notations can be followed in UML and what are their limitations. First, we captured relevant parts of JavaPhone using architectural notations

adequate for each of the classes of views above. Then, we explored alternative UML representations for each of these views, and discussed the usefulness of such documentation with specialists from DaimlerChrysler. We paid special attention to run-time behavioral views, for which we gave a flavor of two languages for architecture description that are directly aimed at expressing architectural concerns (Acme and Wright).

The primary outcome of this discussion is that it is possible to improve current practices of architectural description using notations like the ones we described, and by separating architectural concerns into separate views. We looked at ways of using UML to provide documentation for the views, as well as at the semantic guidelines for interpreting each documented view. In each case, we pointed out the strengths, weaknesses, and pitfalls of using UML-like object notations.

4.4. Architectural design and object-oriented design (UML 2.0)

We have worked with the Object Management Group's (OMG) Analysis and Design Platform Task Force (http://www.celigent.com/omg/adptf/) to draft a Request for Proposals (RFP) for UML, version 2.0. This request for proposals includes a section detailing requirements for enhancements to UML to better support architectural modeling.

In addition we evaluated a set of techniques for using existing UML to model software architectures [14]. It provides an extensive comparison of alternative modeling strategies, and evaluates the strengths and weaknesses of each.

5. Conclusions and future work

The investigations conducted in the process of this grant have aimed to reduce the cost of software change by allowing designs to be captured in the first place, and by providing tool support to enforce these designs as the software is changed. To capture the design of the system, we have developed the architectural description languages Acme, UniCon and Wright, with associated tool support, to allow designs to be specified and analyzed. These tools and languages have been used to analyze realistic specifications of software, and have been useful in uncovering design faults before those designs have been translated into implementations. This has had a direct saving on software development for these projects. In particular, the discovery of numerous errors in the HLA specification saved a considerable amount of time and money, had the design been implemented prior to analysis using our techniques.

Although we have made significant progress in tool support for software architecture, there are several areas of research that must still be conducted. The first of these is the relationship between architectural design and other design techniques. As mentioned in Section 4.4, we have begun collaboration with the UML community to clarify this relationship.

Although the tools and techniques we have developed are suitable for capturing designs, we have only just begun investigating the correspondence between the design of a software system and its implementation. Two areas of research where this

correspondence will apply are in being able to verify that an implementation satisfies the software design, and in situations where the software is required to evolve at runtime. The latter situation is an area in which we are particularly interested.

6. References

[1] R. Allen. *A Formal Approach to Software Architecture.* Ph.D. Thesis, published as Technical Report CMU-CS-97-144, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA. May 1997.

[2] R. Allen and D. Garlan. *A Formal Basis for Architectural Connection.* ACM Transactions on Software Engineering and Methodology. July 1997.

[3] P. Binns and S. Vestal. *Formal real-time architecture specification and analysis.* In Proceedings of the Tenth IEEE Workshop on Real-Time Operating Systems and Software, New York, NY, May 1993.

[4] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. *Domain-Specific Software Architectures for Guidance, Navigation, and Control.* International Journal of Software Engineering and Knowledge Engineering, January 1994.

[5] M. Burrows, M. Abadi and R. Needham. *A Logic of Authentication.* Technical Report SRC-39, DEC SRC, 1989.

[6] C. Damon. *Selective Enumeration.* Ph.D. Thesis, Technical Report CMU-CS-00-151, Department of Computer Science, Carnegie Mellon University, July 2000.

[7] R. DeLine. *Avoiding packaging mismatch with Flexible Packaging.* In ICSE'99, Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, CA, pp. 97-106. May 1999.

[8] R. DeLine. *Resolving Architectural Mismatch.* Ph.D. Thesis, Carnegie Mellon University School of Computer Science, 1999.

[9] L.G. DeMichiel, L.Ü. Yalçinalp and S. Krishnan. *Enterprise JavaBeans™ Specification, Version 2.0.* Sun Microsystems, October 2000.

[10] Formal Systems (Europe) Ltd. *Failures Divergence Refinement: User Manual and Tutorial.* Oxford, England, 1.2β edition, October 1992.

[11] D. Garlan, R. Allen and J. Ockerbloom. *Architectural Mismatch, or Why it's hard to build systems out of existing parts.* Proceedings of the 17th International Conference on Software Engineering (ICSE-17), April 1995.

[12] D. Garlan, R.T. Monroe and D. Wile. Acme: Architectural Description of Component-Based Systems, in Foundations of Component-Based Systems, G.T. Leavens and Murali Sitaraman (Eds), Cambridge University Press pp. 47—68, January 2000.

[13] D. Garlan, A. Kompanek, J. Kenney, D. Luckham, B. Schmerl and D. Wile. *An Activity Language for the ADL Toolkit.* Draft proposal.

[14] D. Garlan an A. Kompanek. *Reconciling the Needs of Architectural Description with Object-Modeling Notations.* Proceedings of the Third International Conference on the Unified Modeling Language - << UML >> 2000, York, UK, October 2000.

[15] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, N.J. 1985.

[16] J. Hu. *Adding Maps to Acme.* Draft proposal.

[17] R. Kailar. *Accountability in electronic commerce protocols.* IEEE Transactions on Software Engineering **25**(5):313-328, May 1996.

[18] D. Kindred. *Theory Generation for Security Protocols.* Ph.D. Thesis, Technical Report CMU-CS-99-130, Department of Computer Science, Carnegie Mellon University, May 1999.

[19] P. Kruchten. *The 4+1 View Model of Architecture.* IEEE Software, November 1995.

[20] D.C. Luckham, L.M. Augustin, J.J. Kenney, J. Veera, D. Bryan, and W. Mann. *Specification and analysis of system architectures using Rapide.* IEEE Transactions on Software Engineering, Special Issue on Software Architecture, **21**(4):336—355, April 1995.

[21] N. Medvidovic and R. Taylor. *A Framework for Classifying and Comparing Architecture Description Languages.* In Proceedings of European Software Engineering Conference 1997, September 1997.

[22] R. Monroe. *Rapid Development of Custom Software Architecture Design Environments.* Ph.D. Thesis, published as Technical Report CMU-CS-99-161, Carnegie Mellon University School of Computer Science, August 1999.

[23] R. Monroe. *Capturing Software Architecture Design Expertise with Armani.* Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, 1998.

[24] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools.* Ph.D. Thesis, Carnegie Mellon School of Computer Science. To be published.

[25] J.P. Sousa and D. Garlan. *Formal Modeling of the Enterprise JavaBeans Component Integration Framework,* in Information and Software Technology Special Issue on Component Based Development **42**(14), Elsevier Print, UK, November 2000.

[26] Sun Microsystems. *The JavaPhone™ 1.0 API Specification.* Available at http://java.sun.com/products/javaphone/download.html. October 2000.

DISTRIBUTION LIST

addresses	number of copies
DR. ROY F. STRATTON AFRL/IFTD 525 BROOKS ROAD ROME, NY 13441-4505	5
CARNEGIE MELLON UNIVERSITY SCHOOL OF COMPUTER SCIENCES 5000 FORBES AVENUE PITTSBURGH, PA 15213	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
AFRL/IFT 525 BROOKS ROAD ROME, NY 13441-4505	1
AFRL/IFTM 525 BROOKS ROAD ROME, NY 13441-4505	1

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.